Preprint
UCRL-JC-148310

Local Discovery of System Architecture – Application Parameter Sensitivity: An Empirical Technique for Adaptive Grid Applications

I.R. Corey, J.R. Johnson, J.S. Vetter

This article was submitted to
The 11th IEEE International Symposium on High Performance Distributed Computing (HPDC11), Edinburgh, Scotland, July 24-26, 2002

U.S. Department of Energy
Lawrence Livermore National Laboratory

May 8, 2002

Local Discovery of System Architecture – Application Parameter Sensitivity: An Empirical Technique for Adaptive Grid Applications*

I.R. Corey, J.R. Johnson, J.S. Vetter
Computing Applications & Research Department
Lawrence Livermore National Laboratory

Abstract

This study presents a technique that can significantly improve the performance of a distributed application by allowing the application to locally adapt to architectural characteristics of distinct resources in a distributed system. Application performance is sensitive to *system architecture– application parameter* pairings. In a distributed or Grid enabled application, a single parameter configuration for the whole application will not always be optimal for every participating resource. In particular, some configurations can significantly degrade performance. Furthermore, the behavior of a system may change during the course of the run. The technique described here provides an automated mechanism for run-time adaptation of application parameters to the local system architecture. Using a scaled-down simulation of a Monte Carlo physics code, we demonstrate that this technique can conservatively achieve speedups up to 65% on individual resources and may even provide order of magnitude speedup in the extreme case.

1. Introduction

The interaction between the parameterization of an application and the architectural characteristics of the system on which the application is running is of fundamental importance to the overall performance of the application. For a distributed application, performance inequities between different resources due to favorable and unfavorable *application parameter – system architecture* pairings can significantly affect overall performance.

There is currently no general theoretical model for predicting performance based on parameter – architecture pairings. However there has been success with using empirical methods, ([1], [5], [9], [10], [13]). These empirical systems perform extensive testing of a range of parameter values when they are installed on distinct architectures. In a heterogeneous, distributed environment, performing the empirical test at installation time is neither practical nor is it always possible. A single parameter configuration for the whole application will not always be optimal for every resource participating in a distributed computation. Resource characteristics may not be known at scheduling time and resources may come and go throughout the computation. In addition, executables may be staged rather than existing as highly tuned software already installed on a resource.

By using a combination of micro benchmarking, performance assertions and sensitivity analysis, the technique presented here allows a locally scheduled application component to adapt at run-time to the resource to which it has been assigned and then continue monitoring its performance and adapt as necessary. This technique is an extension of earlier empirical studies in that it operates on a large-scale application and is applied at run-time rather than install-time.

The technique works as follows, (Figure 1): The application code is instrumented with performance assertions at positions in the code where system and problem parameters can be dynamically varied, (e.g. between time-steps, or modes such as grid generation and time dependent simulation).

When an instance of the application is deployed on a local resource in a distributed environment, it first runs a micro benchmark varying problem parameters one at time and finds the parameters that have the greatest affect on performance. This micro benchmark can either be performed within the application itself or by a separate benchmarking tool included as a library to the application.

From the aggregate benchmark data the best parameter configuration for the local resource is determined and the application is modified to use this

* This work was performed under the auspices of the U. S. Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

configuration. In addition, performance assertion bounds and handlers are set using the micro benchmark results.

After the micro benchmark phase is complete and the configured application is running, if the code fails to meet a performance assertion based on the original micro benchmark, then control is transferred to a function registered with the performance assertion that decides whether or not to modify a parameter value. Should the function determine that it is necessary to vary a parameter value, then additional benchmarking is performed beginning with the parameters to which performance is most sensitive. A new configuration is constructed and the application is modified to reflect the changes in the system behavior.

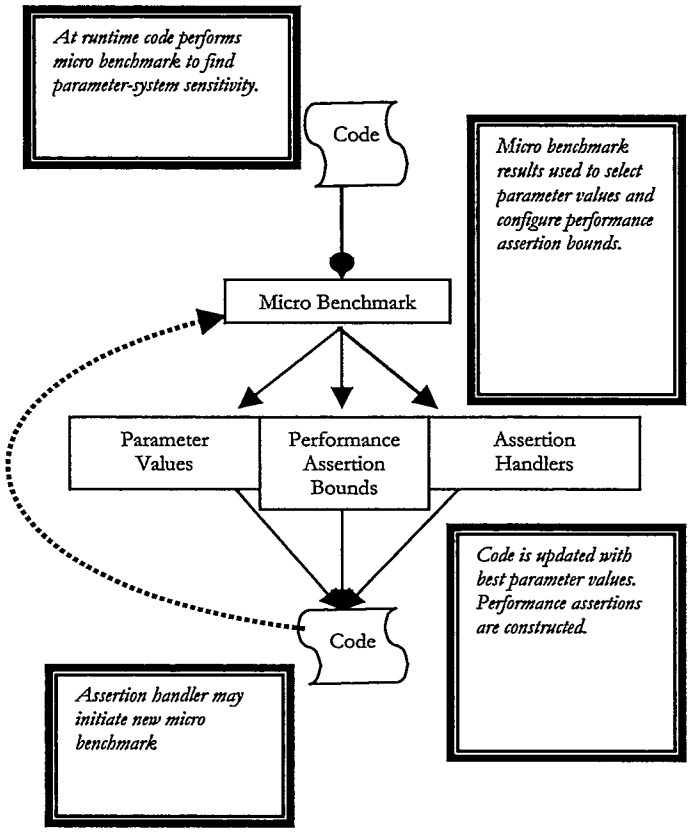


Figure 1. Illustration of Adaptive Application Technique

2. Related Work

An initial attempt at a formal framework for parameterized architecture adaptive algorithms was described by Ueberhuber and Krommer, ([8]). More recently, tangible empirical results on homogeneous, static architectures have been presented in ([1], [5], [9], [13]). NetSolve, ([2]), is a network tool designed to map a problem to the best available resource in a distributed environment based on network performance. NWS, ([14]), and Globus ([4], http://www.globus.org) provide information on the status of networks and the availability of servers. The GrADS project (http://nhse2.cs.rice.edu/grads/) is addressing issues of application performance and performance contracts on computational grids. Recently the GrADS project has presented a framework for adaptive Grid programs, ([7]). Code instrumentation and dynamic steering is explored in ([6], [11]).

3. Monte Carlo Simulation

The simulation used here is designed to emulate a large-scale Monte Carlo physics code under development at LLNL. The simulation models the computation-communication structure of the Monte Carlo code, but does not perform any of the physics. It is primarily used for rapid prototyping and testing new communication ideas before they are fully implemented in the larger Monte Carlo code. Both the actual code and the simulation are structured in a *communicate, work, reduce* cycle. This iterative structure simplifies the adaptive process by allowing the adaptation to take place between iterations.

For this experiment only communication parameters were investigated. Both the simulation and the Monte Carlo code have been constructed so that it is easy to dynamically change communication and MPI parameters (e.g. type of send, buffer size, etc...).

4. Micro Benchmark Description

For the results described here three separate experiments were run varying, *message size, type of send, "message factor"* and *"send buffer."*

For the first set of experiments, *message size* was held constant at 1 MB. In the second set, it ranged between 1, 2, 4 and 8 MB. In the third set, it ranged between 64, 128, 256, and 512 KB. *Type of send* was varied between *Irsend, Isend, Issend. Message factor,* is the number of equally sized units by which the message is divided. For example, with a 1MB message, a *message factor* of 2 sends 2 512K messages. A *message factor* of 4 sends 4 256K messages, etc. The results from all experiments presented here use *message factors* of 1, 2, 4 and 8. *Send buffer* is the number of sends held in the buffer before waiting. The results presented here use *send buffers* of 0, 4, 7, 10, in the first experiment and 0, 32, 64, and 96 in the second and third. Holding all other parameters constant and just varying these four one at a time generated a total of 48 tests (4*4*3) per run on each of 2 architecture in the first experiment and 192

(3*4*4*4) tests on each of 3 architectures in both the second and third experiments.

The first of these tests were run on two distinct architectures: 3 nodes/2 processors per node of IBM's ASCI Blue Pacific and 4 processors on a single node on LLNL's GPS Cluster (Compaq ES45 with 4 1GHz EV6.8 processors). The second and third group of tests were run on 2 nodes / 2 processors per node of IBM's ASCI Blue Pacific, 2 nodes / 2 processors of IBM's ASCI White and 4 processors on a single node of LLNL's Tera Cluster which consists of 32 Compaq AlphaServer and DS20 systems. For each test we ran the code for 100 iterations and measured wall-clock time, MPI-time and throughput.

5. Results

Results described here are from multiple runs of the micro benchmark using the simulation code.

The first thing the results show is that each architecture has a distinctive 'signature' that can be seen by plotting total application time as a function of the application parameters, (Figure 2, Figure 3). The x-axis in these figures is an index into the parameterization, while the y-axis is the total run-time associated with the parameterization. The sort order for the x-axis is: *(send type, message factor, send buffer, message size)*. For example, the first four points on each of the graphs in Figure 2, correspond to the following parameterizations *(Irsend, 1, 0, 1)*, *(Irsend, 1, 0, 2)*, *(Irsend, 1, 0, 4)* and *(Irsend, 1, 0, 8)*. At the 5th point, send buffer varies from 0 to 32. At the 17th point, message factor varies from 1 to 2, and at the 65th point send type varies from Irsend to Isend.

While these signature capture the distinction between architectures, similarities can also be seen with the greatest similarity between ASCI Blue and ASCI White. In all cases, send type stands out as a distinctive feature and forms the 3 prominent groupings seen in each of the figures. For a given send type in the second series of experiments, message factor and message size are the dominating performance features on the ASCI machines with message factor = 1 outperforming 2, 4 and 8 in the second series of experiments, (Table 1). For the third set, message size dominates performance. On the Compaq machines, message size also has a more noticeable impact within send type.

Drilling down into the data makes these observations clear and shows that although ASCI Blue and White have similar profiles, there are important performance differentiators between these two architectures.

Type of Send	*MF*	*Max (s)*	*Min (s)*	*Ave (s)*	*Std Dev*
Irsend	1	71.95	58.255	64.34667	6.187918
	2	101.03	96.315	98.12844	1.824651
	4	101.09	96.35	98.18438	1.826172
	8	101.32	77.485	90.46846	9.677284
Isend	1	71.93	65.865	69.28375	1.790971
	2	100.765	86.972	97.85169	3.301731
	4	100.87	93.92	98.555	2.095307
	8	101.085	91.9	97.69113	3.043057
Issend	1	73.835	65.91	69.83269	2.335991
	2	100.775	89.42	97.33844	3.008229
	4	100.805	91.93	96.95344	3.043722
	8	102.225	93.94	98.2425	2.473233

Table 1. ASCI White message factor performance per send type.

5.1 ASCI White & Blue

From the signature graphs, one can clearly see the three major groupings due to send type (Irsend, Isend, Issend), (Figure 2). Within each of these groups, the lower cluster of points are message factor = 1. An important distinction between White and Blue can be seen in the White data for Irsend, message factor 8, (end of the first grouping in Figure 2), which shows a stronger sensitivity to message size than Blue has. Another distinction can be seen within the message factor grouping. On White the points on the bottom of the grouping (i.e. the best performing) all correspond to message size of 1, while those along the top of the grouping (i.e. the worst performing) all correspond to message size of 8. In contrast, on Blue, the best and worse performing message sizes per message factor differ per message factor. For message factor >1 they are the same as White, however for message factor = 1, they are obtained at message size 4 and 2 respectively. On both White and Blue, send buffer size has almost no effect.

From this we can see that changing message factor from 8 or 4 to 1 on White while keeping all other parameters constant can improve performance by a significant amount, (Table 2).

Type of Send	*Size MB*	*MF*	*Send buffer*	*Wall Clock(s)*
Irsend	8	1	64	71.95
Irsend	8	8	64	101.32
Speedup				**43%**
Irsend	1	1	0	58.25
Irsend	1	4	0	96.42
Speedup				**65%**

Table 2. ASCI White speedup by changing adapting message factor

Similarly, varying message factor on Blue can produce significant speedup, (Table 3).

Type of Send	*Size MB*	*MF*	*Send buffer*	*Wall Clock(s)*
Irsend	8	1	0	113.88
Irsend	8	8	0	157.55
Speedup				**38%**
Isend	4	1	64	104.325
Isend	4	8	64	147.125
Speedup				**41%**

Table 3. ASCI Blue speedup by adapting message factor

Within a single message factor grouping, a change in message size can also have a noticeable effect, (Table 4).

Type of Send	*Size MB*	*MF*	*Send buffer*	*Wall Clock(s)*
Irsend	1	2	0	138.15
Irsend	8	2	0	156.57
Speedup				**13%**
Isend	1	2	0	137.8
Isend	8	2	0	154.775
Speedup				**12%**

Table 4. ASCI Blue speedup by adapting message size

Results on ASCI White from the third series of experiments varying message size from 64K to 512K can be seen in Figure 4. This also offers a distinctive signature. In contrast to the first series, message size rather than message factor is the dominant factor for performance. Varying send type does not offer variation in performance. However, as in the first set of tests, (Figures 2 and 3), message factor does contribute significantly to performance, (Table 5). Contrasting the results for message sizes greater than 1MB, message factors of 1 and 8 perform better than message factors of 2 and 4 for all problem sizes tested in the range 64K – 512K. Once again, send buffer has no noticeable effect on performance.

In the third series of experiments, ASCI Blue results show a very similar pattern to ASCI White, (Figure 5), with message size being the primary performance driver but message factor making a significant difference for a given message size. (For Figures 4 and 5, parameter index sort order is *send type, message size, message factor, send buffer*). An interesting distinction between White and Blue is in the case of message factor 4. On ASCI Blue, a message factor of 4 was consistently the worst for a given send type and a given message size, whereas on ASCI White, message factor 2 was the worst for each send type and message size.

Type of Send	*Size KB*	*MF*	*Send buffer*	*Wall Clock(s)*
Irsend	512	1	32	30.385
Irsend	512	2	32	49.91
Speedup				**64%**
Isend	512	1	32	35.23
Isend	512	2	32	49.865
Speedup				**41%**
Issend	512	1	32	34.435
Issend	512	2	32	42.07
Speedup				**22%**
Isend	512	2	32	49.91
Isend	512	8	32	42.04
Speedup				**24%**

Table 5. ASCI White speedup by adapting message factor

5.2 Compaq

The results on the Compaq machines are not as separable as those on the IBM ASCI machines. However, discernable patterns are still evident. Like the ASCI machines, there is an observable distinction

between send type, more noticeable in the signature graph from the first experiment with the extreme values removed (Figure 3). Message size is a stronger differentiator than message factor. For Issend, a message size of 4MB consistently performed the best. However, for Irsend, 4MB almost always performed worst, but 1MB message size consistently performed the best. For Isend, 8MB was the best message size, (Table 6).

Type of Send	*Size MB*	*Max (s)*	*Min (s)*	*Ave (s)*
Irsend	1	168.65	127.846	140.98
Irsend	4	1196	139.729	614.157
Issend	4	164.483	149.004	154.31
Isend	8	220.842	166.733	184.09

Table 6. Compaq Tera Cluster performance as a factor of message size

The third set of tests with smaller message sizes also shows sensitivity to both send type and message size, (Figure 6). For a 64K message, with message factor 1 and send buffer 0, Isend performed remarkably better than other send types, (Table 7). There is also a significant speedup from the fastest Irsend to Isend.

Type of Send	*Size KB*	*MF*	*Send buffer*	*Wall Clock(s)*
Irsend	64	1	0	42.946
Isend	64	1	0	7.158
Speedup				**6x**
Issend	64	1	0	27.029
Isend	64	1	0	7.258
Speedup				**58%**
Irsend	64	1	0	16.752
Isend	64	1	0	7.158
Speedup				**>2x**

Table 7. Compaq Tera Cluster speedup as a factor of send type

The horizontal lines in Figure 6 correspond to distinct message sizes and the stability in these regions clearly show the sensitivity to message size. One interesting distinction is for Isend, message size 256K, (points 97 – 112 along the x-axis of Figure 5). The higher horizontal line in this grouping corresponds to message factors of 1 and 2, while the lower line corresponds to message factors 4 and 8 with a 20% performance jump from the fastest in the message factor 1 or 2 group and the slowest in the message factor 4 or 8 group.

The results from the smaller GPS profile show a similar sensitivity for *send-type* but *send buffer* had a more significant effect on performance with all instances of send buffer=0 significantly outperforming tests with send buffer > 0, (Table 8).

Send buffer	*Max (s)*	*Min (s)*
0	216.121	144.858
> 0	525.888	325.558
***Minimum* Speedup**		**50%**

Table 8.GPS speedup adapting send buffer size

For each of the architectures, the difference between the worst parameterization and the best parameterization is quite significant, (Table 9)

Architecture	*Worst Params Performance (s)*	*Best Params Performance (s)*	*Speedup*
ASCI White	102.225	58.255	75%
ASCI Blue	158.455	104	52%
Tera	2034.321	100.521	20x
GPS	525.888	144.858	3.6x

Table 9 Speedup achieved by replacing worst parameterization with best parameterization

6. Conclusions

These results show that for a given application, different architectures exhibit sensitivity to different parameters in a program's configuration. By changing the appropriate parameters for the given architecture, performance can be significantly improved. In the simple simulation studied here, a conservative speedup of from 12 % to 65% on a single resource is achieved.

While a good configuration for a particular architecture can often be discovered analytically, the complexity of building a rules system to handle every possible scenario can be unwieldy. Sometimes results are unexpected but not unreasonable. This complexity and the fact that in a heterogeneous distributed computing environment, resource characteristics may not be known even at run-time lead to the conclusion that an empirical approach is the best for dynamic application

configuration. The simulation described here shows that running application-specific micro benchmarks on the actual system offers a good mechanism for determining the effect of parameter-architecture pairings and finding parameters to which the architecture is most sensitive thus allowing the application to run using a good configuration on the local system.

For complex codes, a stand-alone micro benchmark such as the simulation described here is a good tool for discovering the behavior of a system. For codes that have clear delineation of behavior (e.g. a computation phase followed by a communication phase), the code itself may be used to perform the initial micro-benchmark.

One of the challenges in evaluating micro benchmarks is choosing a metric to evaluate the system. Wall clock time is not always the best metric. This difficulty can be addressed by including data from system utilities such as hardware performance counters in the benchmarking phase. This benchmark data can be used to construct performance assertions that are inserted between time-steps to determine whether the behavior of the system is consistent with the original micro benchmark. If not, the performance assertion will execute a handler function to rerun the micro benchmark and re-configure the application if necessary.

Performance assertions [12] are an important component of this technique since they allow the application to continually monitor its operation and adapt as necessary. Performance assertions measure empirical performance data for individual operations within the application and if those measurements fail some user-defined expectation, they react using a variety of methods. The performance expectation is an expression that compares the measurements using a relational operator. The expression can include measured performance data, architectural specific data (e.g., peak rate), micro benchmark data (e.g., sustained rate), constants, and various operators. The method that we use here invokes an application subroutine to affect a change in the application. A real-world application may change its behavior over the course of the run and a different configuration may be optimal. As an example take the profile for ASCI White Irsend. If the program were to change its message size from 8MB to 1MB and the current message factor were 8, there would be a noticeable degradation in performance. For this example, performance assertions would invoke a user-defined subroutine that modifies the parameters to which the particular architecture is most sensitive: in this case, message factor or send type. For either case, performance would improve. In this way, performance assertions assist the application in adapting to its local resource throughout the course of the run.

The micro benchmark is essential in building the performance assertions and constructing the handler functions that adapt the application while it is running. In the results presented here, the micro benchmark shows that send buffer does not have any appreciable effect on performance for the IBM machines, however it does on the Compaq GPS. Thus when adapting the application on the IBM machines, message size and message factors are the parameters that are modified first to improve performance, while on the Compaq, send buffer may be retained as a parameter to vary. Similarly, on the IBM machines, varying send type does not have as direct an effect on performance as does message size and message factor, but it does show some performance variation on the Compaq Tera cluster. The results of the micro benchmark yield a list of parameters ordered by sensitivity which allows the adaptation phase to start by manipulating the parameters to which the application – architecture pairing is most sensitive and essentially ignore (unless everything else fails) the parameters to which it is least sensitive.

7. Future Work

The results presented here are based on a handful of communication parameters. There are many more parameters that can be varied. In addition there are non-communication parameters that may also significantly affect performance (e.g. threads, problem size, array strides and blocking etc...). Choosing the best parameters to use in the micro benchmark is an area for further exploration.

Another area that needs additional study is the construction of the adaptation functions registered with the performance assertions. The approach described here is a naïve one that simply starts with the most sensitive parameter and varies it until something improves. If there is no improvement, it switches to the next most sensitive parameter and so on. A more sophisticated sensitivity analysis may offer greater insight and allow the application performance to converge faster. As can be seen in this study, some variations are non-linear (the effect of message factor on the ASCI White and Blue results presented in Figure 4) or singular (sensitivity of ASCI White to problem size only for message factor 8, Irsend in Figure 2) so finding a good sensitivity model that can guide the adaptation phase is an important area for further research.

Extending this study by including more exotic architectures might be fruitful. The fact that such similar architectures as those used here can have such distinct profiles illustrates the usefulness of this technique, but comparing sharply contrasted architectures may lead to additional insight.

The Monte Carlo physics code has been instrumented and tests are being run to see how the full application results compare to the simulation results. The next step after this is to implement this study in a fully distributed testbed to explore how the local adaptation contributes to overall performance improvement.

8. Bibliography

[1] J. Bilmes, K. Asanovic, C-W. Chin, J. Demmel, "Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology.", *International Conference on Supercomputing*, 1997.

[2] H. Casanova, J. Dongarra. "NetSolve; A Network Server for Solving Computational Science Problems.", *International Journal for Supercomputer Applications and High Performance Computing*, vol. 11, no. 3, 1997.

[3] I. Foster, C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann, San Francisco, 1998.

[4] I. Foster, C. Kesselman, Globus: "A Metacomputing Instrastructure Toolkit." *International Journal of Supercomputer Applciations*, vol. 11, no. 2, 1997.

[5] M. Frigo and S. G. Johnson, "FFTW: An Adaptive Software Architecture For The FFT." *ICASSP*, vol. 3, 1998.

[6] J. K. Hollingsworkth, P. Keleher, "Prediction and Adaptation in Active Harmony", *Cluster Computing*, vol. 2, 1999.

[7] K. Kennedy, M. Mazina, et. al., "Toward a Framework for Preparing and Executing Adaptive Grid Programs", *International Parallel and Distributed Processing Symposium.* 2002. (to appear)

[8] A. R. Krommer, C.W. Ueberhuber, "Architecture Adaptive Algorithms", *Parallel Computing*, vol. 19, 1993.

[9] D. Mirkovic, S. L. Johnsson, "Automatic Performance Tuning in the UHFFT Library", *International Conference on Parallel Computing*, 2001.

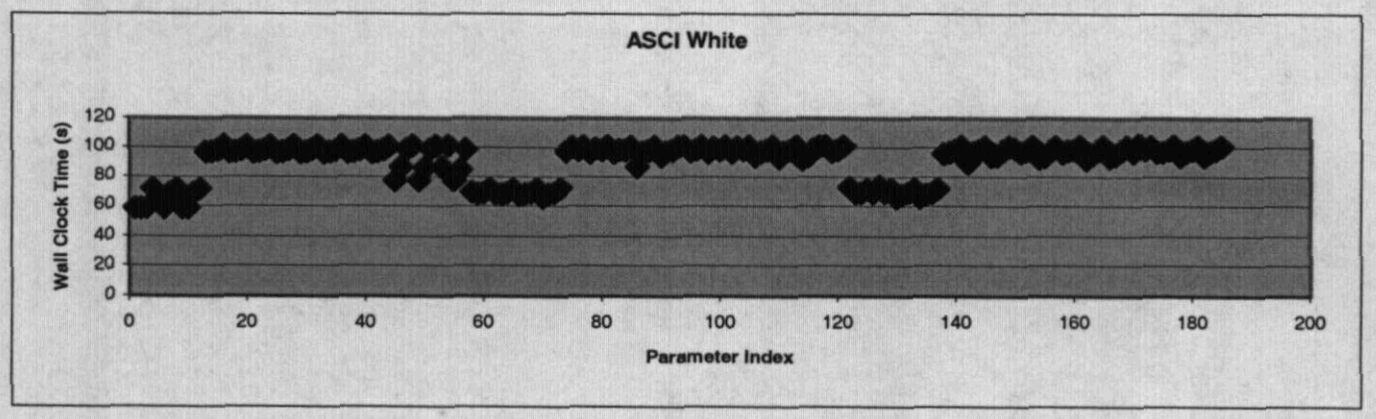
[10] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications", *Supercomputing*, 2000.

[11] J. S. Vetter, D. A. Reed, "Real-time Performance Monitoring, Adaptive Control and Interactive Steering of Computational Grids", *The International Journal of High Performance Computing Applications*, vol. 14, no.4, 2000.

[12] J. S. Vetter and P. Worley, "Performance Assertions: a Performance Diagnosis Tool," Submitted., 2002.

[13] R. C. Whaley, A. Petitet, J. J. Dongarra, "Automated Emprical Optimizations of Software and the ATLAS Project." *Parallel Computing*, vol. 27, no1-2, 2001

[14] R. Wolski, N. Spring, J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing." *Journal of Future Generation Computing Systems*, vol.15, no. 5-6, 1999.



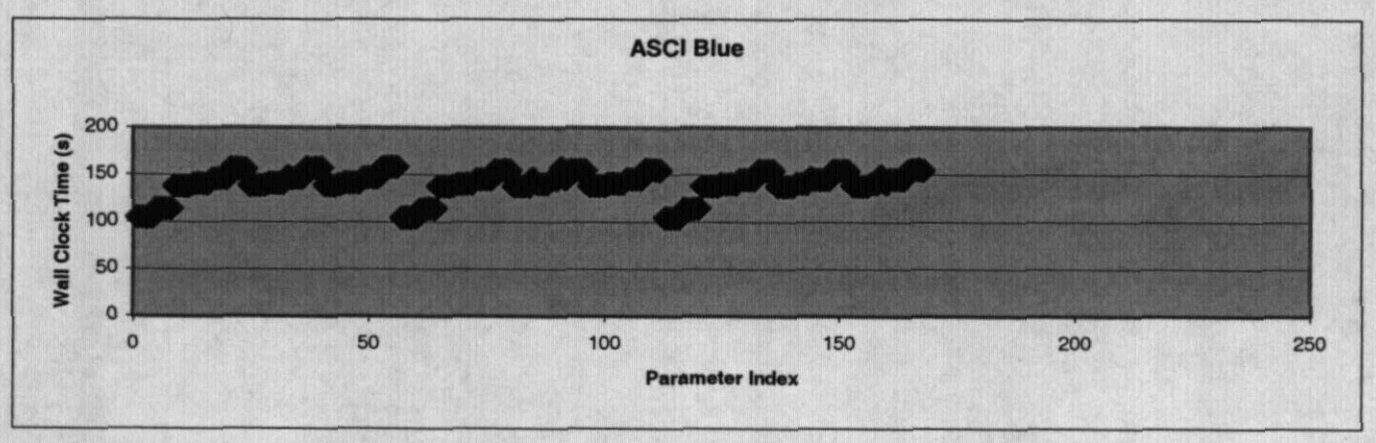
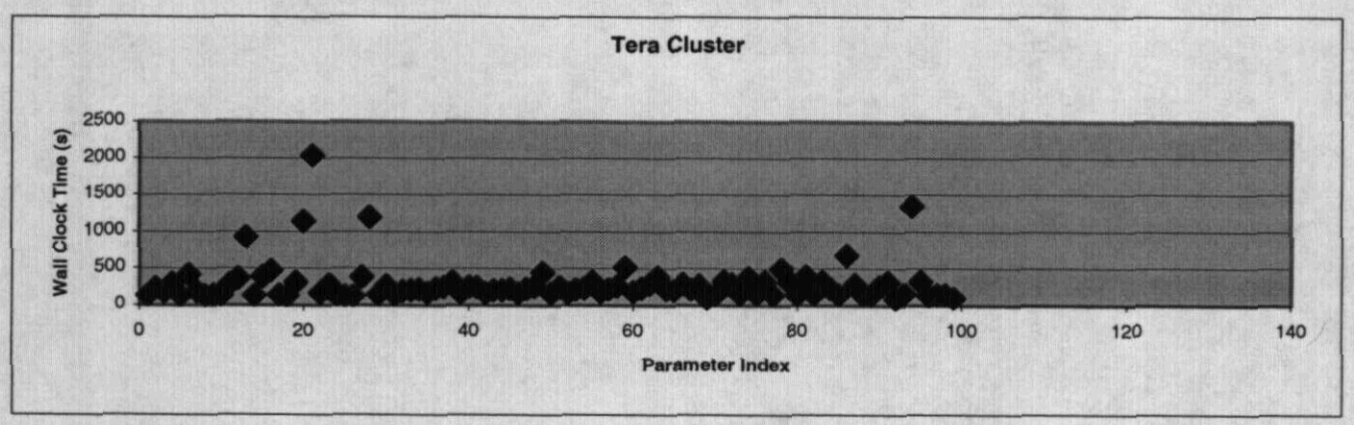
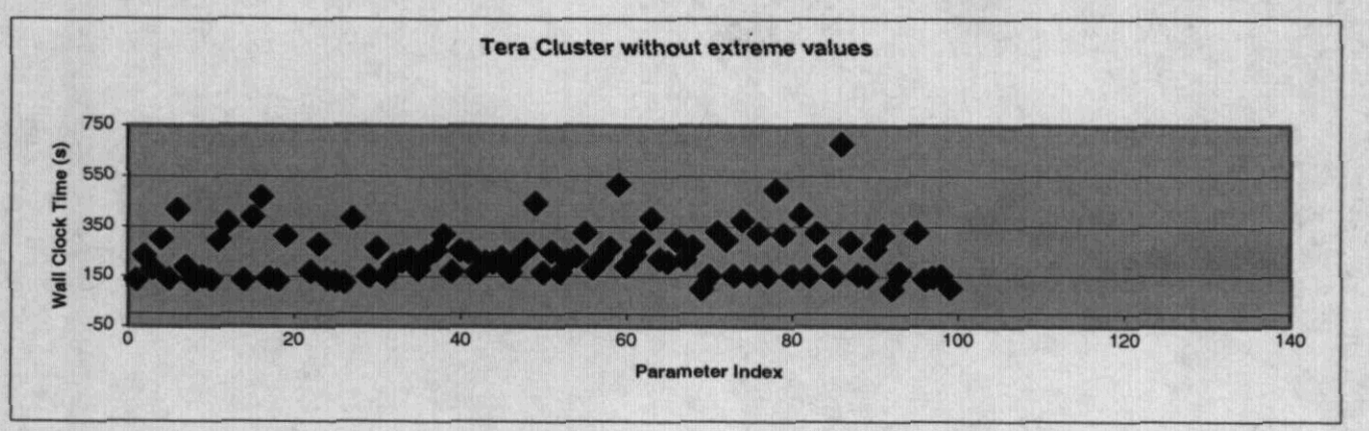


Figure 2. ASCI White & Blue (1MB – 8MB)





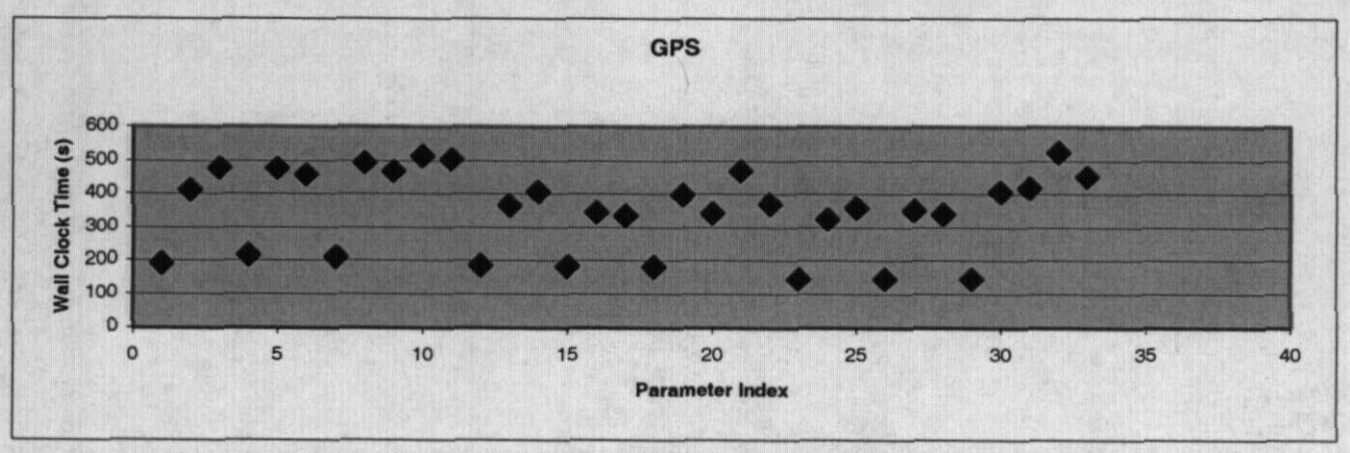


Figure 3. Tera Cluster (1MB – 8MB) and GPS

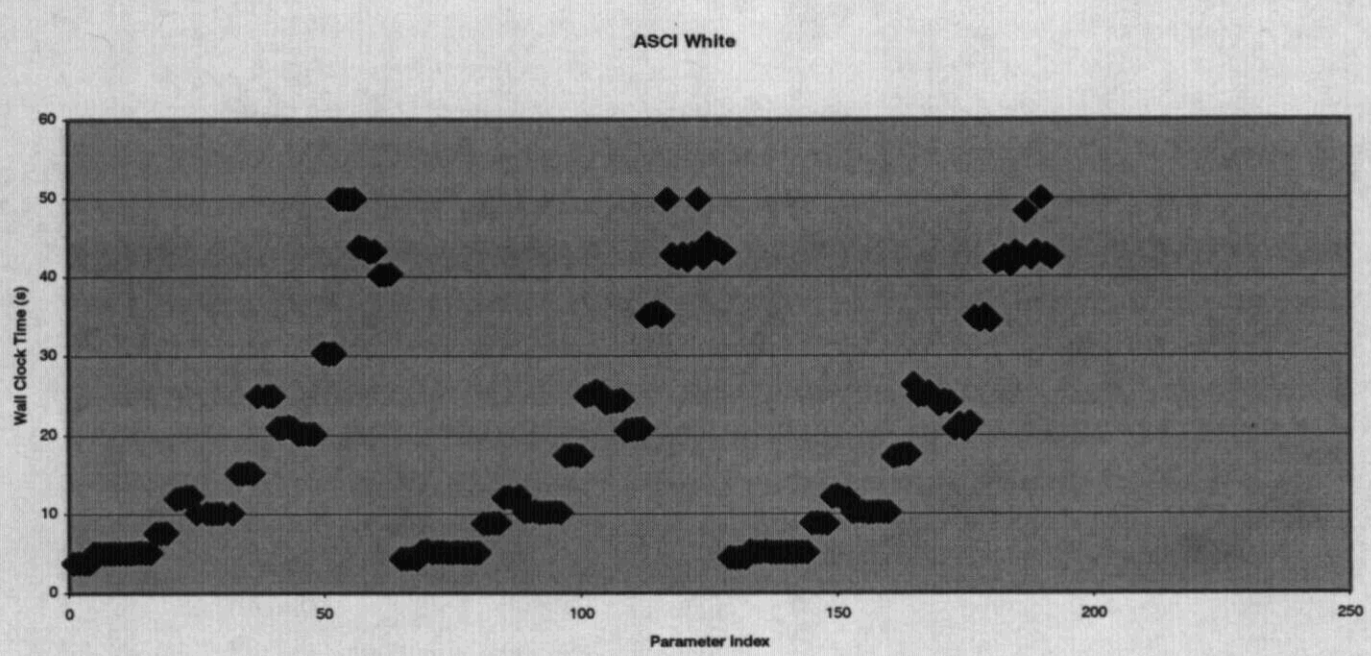


Figure 4. ASCI White (64K – 512K)

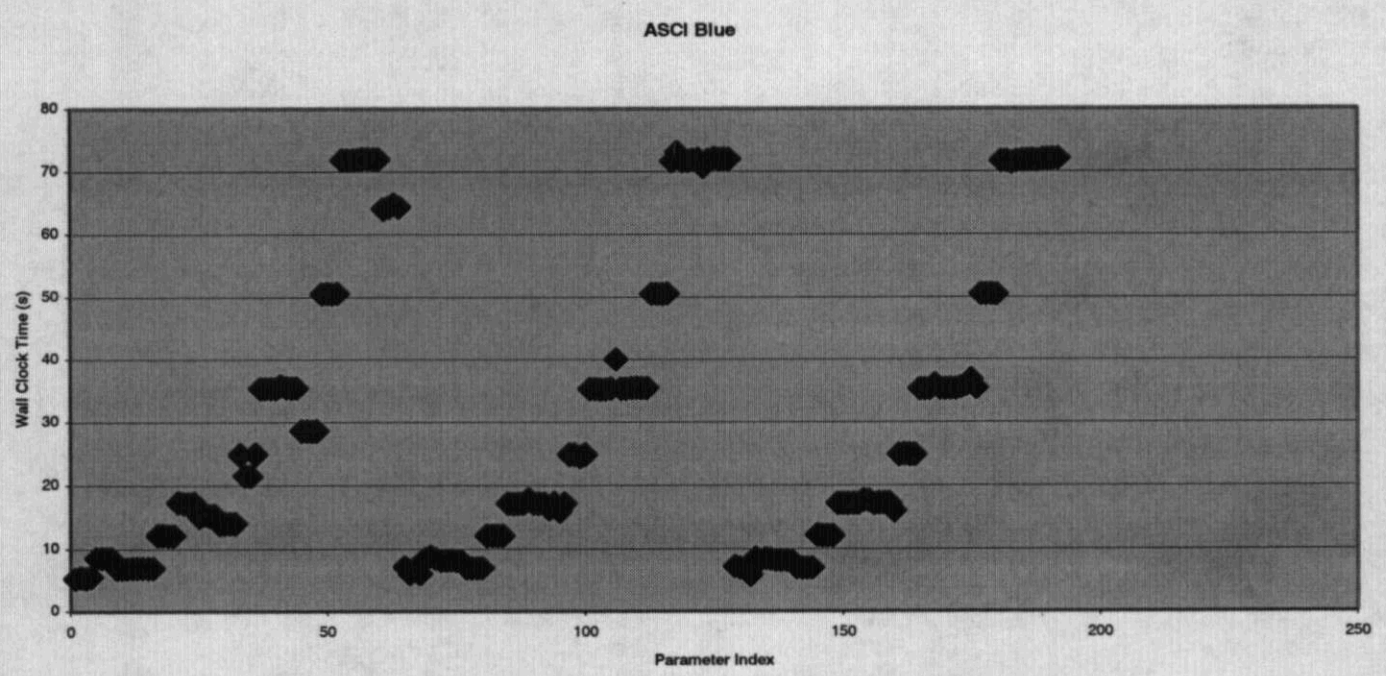


Figure 5. ASCI Blue (64K – 512K)

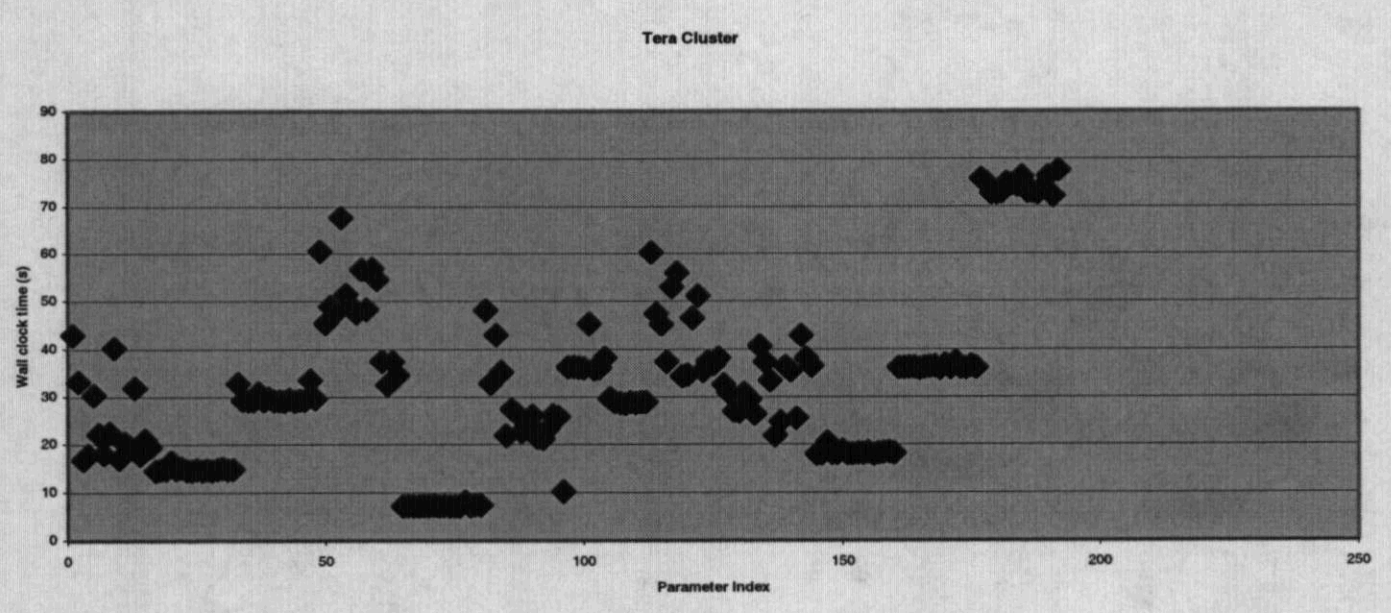


Figure 6. Tera Cluster (64K – 512K)